Parallel program/component adaptivity management

Marco Aldinucci, Françoise André, Jérémy Buisson, Sonia Campa, Massimo Coppola, Marco Danelutto, Corrado Zoccolo

Parallel program/component adaptivity management

M. Aldinucci[a], F. André[b], J. Buisson[b], S. Campa[c], M. Coppola[a], M. Danelutto[c], C. Zoccolo[c]

[a]Inst. of Information Science and Technologies (ISTI – CNR), V. Moruzzi 1, 56124 Pisa, Italy

[b]University of Rennes I, Avenue du General Leclerc, 35042 Rennes, France

[c]Dept. of Computer Science, University of Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy

Grid computing platforms require to handle dynamic behaviour of computing resources within complex parallel applications. We introduce a formalization of adaptive behaviour that separates the abstract model of the application from the implementation design. We exemplify the abstract adaptation schema on two applications, and we show how two quite different approaches to adaptivity, the ASSIST environment and the AFPAC framework, easily map to this common schema.

1. An Abstract Schema for Adaptation

With the advent of more and more complex and dynamic distributed architectures, such as Computational Grids, growing attention has to be paid to the effects of dynamicity on running programs. Even assuming a perfect initial mapping of an application over the computing resources, choices made can be impaired by many factors: load of the used machines and network available bandwidth may vary, nodes can disappear due to network problems, user requirements may change.

To properly handle all these situations, as well as the implicitly dynamic behaviour of several algorithms, *adaptivity* management code has to be built into the parallel/distributed application. In so doing, a tradeoff must be settled between the complexity of adding dynamicity-handling code to the application and the gain in efficiency we obtain.

The need to handle adaptivity has been already addressed in several projects (AppLeS [5], GrADS [10], PCL [8], ProActive [4]). These works focus on several aspects of reconfiguration, e.g. adaptation techniques (GrADS, PCL, ProActive), strategies to decide reconfigurations (GrADS), and how to modify the application configuration to optimize the running application (AppLes, GrADS, PCL). In these projects concrete problems posed by adaptivity have been faced, but little investigation has been done on common abstractions and methodology [9].

In this work we discuss, at a very high level of abstraction, a general model of the activities we need to perform to handle adaptivity in parallel and distributed programs.

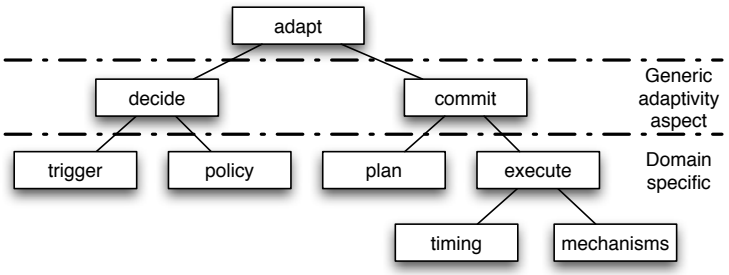

Figure 1: Abstract schema of an adaptation manager.

Our model is abstract with respect to the implemented adaptation techniques, monitoring infrastructure and reconfiguration strategy; in this way we can uncover the common aspects that have to be addressed when developing a programming framework for reconfigurable applications, and we show that it can be applied to two concrete examples: ASSIST [3] and AFPAC [6].

The abstract model of dynamicity management we propose is shown in Fig. 1, where high-level actions rely on lower-level actions and mechanisms. The model is based on the separation of application-oriented abstractions and implementation mechanisms, and is also deliberately specified in minimal way, in order not to introduce details that may constrain possible implementations. As an example, the schema does not impose a strict time ordering among its leaves. In order to validate the proposed abstraction, we exemplify its application in two distinct, significant case studies: message-passing SPMD programs, and component-based, high-level parallel programs. In both cases, adaptive behaviour is derived by specializing the abstract model introduced here. We get significant results on the performance side, thus showing that the model maps to worthwhile and effective implementations [3].

The work is structured as follows. Sec. 2 introduces the abstract model. The various phases required by the general schema are detailed with examples in Sec. 3.1 and Sec. 3.2 with respect to two example applications. Sec. 4 explains how the schema is mapped in the AFPAC framework, where self-adapting code is obtained by semi automated restructuring of existing code. Sec. 5 describes how the same schema is employed in the ASSIST programming environment, exploiting explicit program structure to automatically generate autonomic dynamicity-handling code.

2. Adaptivity

The process of adapting the behaviour of a parallel/distributed application to the dynamic features of the target architecture is built of two distinct phases: a decision phase, and a commit phase, as outlined in Fig. 1. The outcome of the decide phase is an abstract adaptation strategy that the commit phase has to implement. We separate the decisions on the strategy to be used to adapt the application behaviour from the way this strategy is actually performed. The decide phase thus represents an abstraction related to the application structure and behaviour, while commit phase concerns the abstraction of the run-time support needed to adapt. Both phases are split into different items. The decide phase is composed of:

- trigger – It is essentially an interface towards the external world, assessing the need to perform corrective actions. Triggering events can result from various monitoring activities of the platform, from the user requesting a dynamic change at run-time, or from the application itself reacting to some kind of algorithm-related load unbalance.
- policy – It is the part of the decision process where it is chosen how to deal with the triggering event. The aim of the adaptation policy is to find out what behavioural changes are needed, if any, based on the knowledge of the application structure and of its issues. Policies can also differ in the objectives they pursue, e.g. increasing performance, accuracy, fault tolerance, and thus in the triggering events they choose to react to.
  Basic examples of policy are “increase parallelism degree if the application is too slow”, or “reduce parallelism to save resources”. Choosing when to re-balance the load of different parts of the application by redistributing data is a more significant and less obvious policy.

In order to provide the decide phase with a policy, we must identify in the code a pattern of parallel computation, and evaluate possible strategies to improve/adapt the pattern features to the current target architecture. This will result either in specifying a user-defined policy or picking one from a library of policies for common computation patterns. Ideally, the adaptation policy should depend on the chosen pattern and not on its implementation details.

In the commit phase, the decision previously taken is implemented. In order to do that, some assessed plan of execution has to be adopted.

- plan – It states how the decision can be actually implemented, i.e. what list of steps has to be performed to come to the new configuration of the running application, and according to which control flow (total or partial order).
- execute – Once the detailed plan has been devised, the execute phase takes it in charge relying on two kinds of functionalities of the support code
  - the different mechanisms provided by the underlying target architecture, and
  - a timing functionality to activate the elementary steps in the plan, taking into account their control flow and the needed synchronizations among processes/threads in the application.

The actual adapting action depends on both the way the application has been implemented (e.g. message passing or shared memory) and the mechanisms provided by the target architecture to interact with the running application (e.g. adding and removing processes to the application, moving data between processing nodes and so on).

The general schema does not constrain the adaptation handling code to a specific form. It can either consist in library calls, or be template-generated, it can result from instrumenting the application or as a side effect of using explicit code structures/library primitives in writing the application. The approaches clearly differ in the degree of user intervention required to achieve dynamicity.

3. Examples of the abstract decomposition

In order to better explain the abstract adaptation model, we instantiate the model in two different applications, and discuss the meaning that actions and phases in the model assume.

3.1. Task farming

We exemplify the abstract adaptation schema on a task-parallel computation organized around a centralized task scheduler, continuously dispatching works to be performed to the set of available processing elements. For this kind of pattern, both a performance model and a balancing policy are well known, and several different implementation are feasible (e.g. multi-threaded on SMP machines, or processes in a cluster and/or on the Grid). At steady state, maximum efficiency is achieved when the overall service time of the set of processing elements is slightly less than the service time of the dispatcher element.

Triggers are activated, for instance, when (1) the average interarrival time of task incoming is much lower/higher than the service time of the system, (2) on explicit user request to satisfy a new performance contract/level of performance, (3) when built-in monitoring reports increased load on some of the processing elements, even before service time increases too much.

Assuming we care first for computation performance and then resource utilization, the adaptation policy would be like that in Fig. 2. Applying this policy, the decide phase will eventually determine the increase/decrease of a certain magnitude in the allocated computing power, independently of the kind of computing resources.

This decision is passed to the commit phase, where we must produce a detailed plan to implement it (finding/choosing resources, devising a mapping of application processes where appropriate).

Assuming we want to increase the parallelism degree, we will often come up with a simple plan like that in Fig. 3. The given plan is the most usual one, but some steps can be skipped

- when steady state is reached, no configuration change is needed
- if the set of processing elements is slower than the dispatcher, new processing elements should be added to support the computation and reach the steady state
- if the processing elements are much faster than the dispatcher, reduce their number to increase efficiency

Figure 2. A simple farm adaptive policy

1. find a set of available processing elements $\{P_i\}$
2. install code to be executed at the chosen $\{P_i\}$ (i.e. application code, code that interacts with the task scheduler and for dinamicity handling)
3. register with the scheduler all the $\{P_i\}$ for task dispatching
4. inform the monitoring system that new processing element have joined the execution

Figure 3. Plan for increasing resources.

depending on the implementation. For example, a multithreaded program executing on a SMP architecture does not require the code to be installed (step 2). The order may also be different, e.g. swapping steps 3 and 4. Actions listed in the plan exploit mechanisms provided by the implementation, for instance to either fork new threads, or stage and run new processes or even ask for a larger processing time share (on a multiprogrammed system with QoS control at the system level). The list of steps in the plan is also customized w.r.t. application implementation. As an example, whenever computing resources are homogeneous, step 1 is quite simple, while it will require a specific effort to select the best execution plan on heterogeneous resources.

Once the detailed plan has been devised, it has to be executed and its actions have to be orchestrated, choosing proper timing in order that they do not to interfere with each other and with the ongoing computation.

Abstract timing depends on the implementation of the mechanisms, and on the precedence relationship that may be given in the plan. In the given example, steps 1 and 2 can be executed in sequence, but without internal constraint on timing. Step 3 requires a form of synchronization with the scheduler to update its data, or to suspend all the computing elements, depending on actual implementation of the scheduler/worker synchronization. For the same reason, execution of step 4 also may/may not require a restart/update of the monitoring subsystem to take into account the new resources.

3.2. Fast fourier transform

The fast fourier transform can be implemented as a parallel SPMD code which distributes the matrix by lines. It alternates local computation and global matrix transposition steps. A performance model is known that predicts the optimal number of processors for such an application, depending on their power and the cost of communications. The code can thus be made adaptive, by spawning processes when new processors become available. Similarly, when some allocated processors are reclaimed by the operating system, concerned processes have to be safely terminated first. Thanks to the abstract model for dynamic adaptation, such a behavior can be easily designed.

The policy is composed of the following two statements: when the trigger notifies of available processors, and if the optimal number of processors is not overflowed, then the application decides to start new processes; when the trigger notifies that some used processors are reclaimed, some of the processes will be stopped. Given this decision, the commit phase produces a plan. The plans for the two kinds of adaptation are given on Fig. 4 and 5.

This example shows that the implementation mechanisms may depend on several aspects of the application. For example, redistributing a matrix is strongly dependent on the application and its implementation. On the other hand, preparation of the environment may require for example starting daemons (when using LAM-MPI communications), but it is not strictly related to the application code.

1. prepare the environment for the newly recruited processors (start daemons, stage-in files, etc.)
2. spawn processes to be executed by the new processors
3. fix connections between processes such that the new ones can communicate with the others
4. redistribute the matrix in order to balance the load amongst the whole set of processes

Figure 4. Plan for spawning processes.

1. redistribute the matrix in such a way that the terminating processes do not hold any part of the matrix anymore
2. fix connections between processes in order to exclude the processes that are terminating
3. effectively terminate the concerned processes
4. clean everything that has been previously installed specifically for the application

Figure 5. Plan for removing processes.

The mechanisms also impose various constraints on the timing phase of the abstract model, depending on their implementation. This is the case for action 2 of the plan for spawning processes which creates the processes. For an MPI application this action can be implemented either the standard way, with the `MPI_Comm_spawn`, or in an ad-hoc way if the developer requires finer control over process creation. The former approach requires synchronization of already running processes, whereas the latter may not.

4. AFPAC: A generic framework for developers to manage adaptation

The AFPAC framework [6] focuses at present on adaptability of parallel components. Its approach consists in defining the modifications that should be applied to an existing component in order to make it able to adapt itself. Its concrete architecture (Fig.6) can be seen as a specialization of the abstract model of Fig. 1 as follows. Indeed, policy, planner and actions entities implement respectively the policy, plan and mechanisms phases of the abstract model; the timing phase of the abstract model is split over both the executor for handling the control flow and the coordinator for the synchronization with the application. AFPAC does not make appear explicitly the trigger phase as it is considered as an interface, whereas the service entity, modelling the application, has no counter-part in the abstract model. As shown in Fig.6, the decider glues the policy to the external probes in the same way that the decide phase aggregates the trigger and policy phases in the abstract model. The same kind of matching applies between the executor entity and the execute phase.

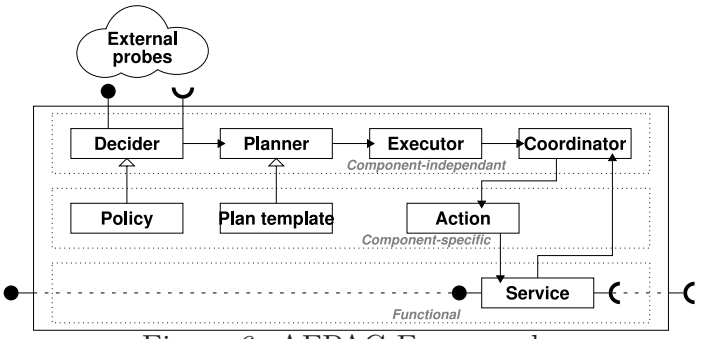

Figure 6: AFPAC Framework.

In the case of a parallel component, the service is implemented by a parallel algorithm. At runtime, it contains several execution threads distributed over a collection of processes. The AFPAC framework does not impose any constraint on communications between threads.

At the current state, the AFPAC framework includes two coordinators. The first one executes sequential actions and does not impose any synchronization constraint with the service. It is somewhat an empty coordinator. The other coordinator aims at executing parallel actions in the context of the execution threads of the service. To do so, it requires to suspend the execution threads at a state from which such actions are allowed to be executed. Such a state is called an adaptation point. In the case of parallel codes, adaptation points must be related in order to build a global state that satisfies some consistency model. For example, in the case of SPMD codes, such a consistency model may state that all threads should execute the action from

the same adaptation point. This problem has been further discussed in [6]; an algorithm has been proposed in [7] for implementing such a coordinator that looks for adaptation points in the future of the execution of the service. It is especially suitable for SPMD codes such as the ones using MPI (e.g. the fast fourier transform example given in Sec.3.2).

The AFPAC framework gives full control over dynamic adaptation to the developer. Consequently, the developer is responsible for designing and implementing the policy, plan template and action entities. In the same way, he/she has to place manually adaptation points within the source code of the service as additional statements. Nevertheless, extra preparation of the component (such as generation of annotations required by the coordinator) is done automatically thanks to aspect-oriented programming. Thanks to this semi-automated modification and to the separation of concerns, AFPAC can be used to make adaptable existing legacy codes at a low development cost.

5. ASSIST: Managing dynamicity using language and compilation approaches

ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of (possibly) parallel modules, interconnected by typed streams of data. A parallel module (*parmod*) coordinates a set of concurrent activities which are performed by *Virtual Processes* (VPs). VPs execute a set of sequential activities on their input data and internal state, activities which are selected on item arrival from the input streams. The sequential functions can be programmed using standard sequential languages (C, C++, Fortran).

Overall, a *parmod* may behave in a data-parallel (e.g. SPMD/for-all/apply-to-all) or task-parallel way (e.g. farm, pipeline), and it can nondeterministically accept from one or more input streams a number of input items, which may be decomposed in parts and used as function parameters to activate VPs. A parmod may also exploit a distributed shared state, which survives between VP activations related to different stream items. More details on the ASSIST environment can be found in [11,2].

An ASSIST module (or a graph of modules) can be declared as a component, which is characterized by provide and use ports (both one-way and RPC-like), and by *Non-Functional* ports. Among the non-functional interfaces there are those related to QoS control.

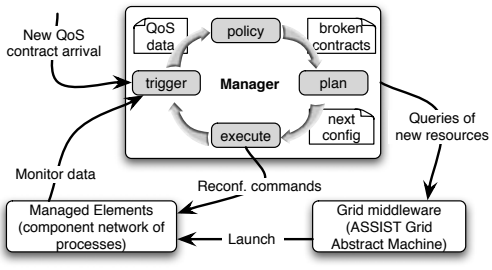

Figure 7: ASSIST framework.

At any moment during an ASSIST application run, components can be assigned a new QoS contract, e.g. specifying a performance requirement. In order to fulfill the contracts, the component framework continuously adapts component configurations, in terms of parallelism degree, and process mapping [3]. The adaptation mechanism relies on automatic user code instrumentation, and on a hierarchy of Application Managers [1].

Each component has a Component Adaptation Manager (CAM) entity coordinating its adaptation. An Application Manager (AM), possibly distributed, enforces QoS of the application in the whole, by coordinating and leveraging on CAMs. As sketched in Fig. 7, ASSIST implements the abstract adaptation schema by organizing its leaves, left to right (compare with Fig. 1) in an autonomic control loop. CAM managed entities are processes within a component, while the AM applies the abstract model to application components. In the following we describe the CAM case.

The trigger functionality has to (1) collect a stream of monitoring data from the running program, as a feedback to the autonomic behaviour of AMs, and (2) to react to QoS contract changes when they trigger the need for adaptation.

The policy phase in Fig. 7 evaluates a component performance model over the monitoring data, to find out the amount/allocation of resources that can match the assigned QoS contract. In the case the QoS contract is broken, the decide phase will set a target for the commit phase, e.g. the additional amount of required computing power. The ASSIST compiler synthesizes the performance model from static information on the parallel pattern exploited by the component. Application programmers can also override standard performance models with custom ones.

The plan phase in Fig. 7 reconveys component performance within its contractually specified values by exploiting the set of actions available as mechanisms. Plan templates are instantiated as partially ordered sets of actions, which are performed according to the schedule provided by timing. ASSIST implements two layers of adaptation mechanisms: parallelism degree management (add or remove resource to/from computation), and computation (VP) remapping, with associated data migration and global state consolidation.

The timing functionality, not shown in Fig. 7, involves a distributed agreement among a set of VPs on the point where the reconfiguration must happen. In ASSIST the migration process can be performed in so-called *reconf-safe* points [3], i.e. points in the application code where the distributed computation and state are known to be consistent, and can be efficiently synchronized. Placement and use of reconf-safe points are automated, so that different mechanisms available to the execute phase (reconfiguration commands in Fig.7) automatically get the appropriate kind of synchronization.

The execute functionality thus exploits support code built within the VPs, and coordinates it with services provided by the component framework to interface to Grid middleware (e.g. for resource recruiting).

Observe that all the code needed to perform the timing and execute phases is automatically generated by the ASSIST compiler, that instruments the application code in a fully transparent manner for the application developer. ASSIST reconf-safe points are designed to exploit synchronization points already needed to ensure the correctness of the parallel application code. Moreover, the ASSIST high-level structured nature enables the compiler to automatically select the optimal implementation of mechanisms for each application and reconf-safe point. For instance, no state migration code is inserted for stateless computations, and depending on the parallelism pattern (e.g. stream versus data parallel), VPs involved in the synchronisation can be a subset of those within the component being reconfigured.

In this way ASSIST adaptive components run with no overhead with respect to non-adaptive versions of the same code, when no configuration change is performed [3].

6. Conclusions

We have described a general model to provide adaptive behaviour in Grid-oriented component-based applications. The general schema we have shown is independent of implementation choices, such as the responsibility for inserting the adaptatation code (either left to the programmer, as it happens in the AFPAC framework, or performed by exploiting knowledge of the high level program structure, as it happens in the ASSIST context). The model also encompasses user-driven as well as autonomic adaptation.

The abstract model helps in separating application and run-time programming concerns of adaptation, exposing adaptive behaviour as an aspect of application programming, formalizing

the concerns to be addressed, and encouraging an abstract view of the run-time mechanisms for dynamic reconfiguration.

This formalization gives the basis for defining a methodology. The given case studies provide with valuable clues about how to solve different concerns, and how to identify common parts of the adaptation that can be generalized in support frameworks. The model can be thus also usefully applied within other programming frameworks, like GrADS, which do not enforce a strong separation of adaptivity issues into design and implementation.

We expect that such a methodology will lead to more portable and understandable adaptive applications and components, and it will also promote layered software architectures for adaptation, simplifying implementation of both the programming framework and the applications.

Acknowledgments. This research work is carried out under the FP6 Network of Excellence *CoreGRID* funded by the European Commission (Contract IST-2002-004265), and it was partially supported by the Italian MIUR FIRB project *Grid.it* (n. RBNE01KNFP) on High-performance Grid platforms and tools.

References

[1] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high performance Grid programming in Grid.it. In V. Getov and T. Kielmann, editors, *Proc. of the Workshop on Component Models and Systems for Grid Applications (June 2004, Saint Malo France)*. Springer, January 2005.

[2] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance Grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer, 2005. (to appear, draft available as TR-04-09, Dept. of Computer Science, University of Pisa, Italy, Feb. 2004).

[3] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005: Parallel and Distributed Computing*, LNCS, Lisboa, Portugal, August 2005. Springer. To appear.

[4] Françoise Baude, Denis Caromel, and Matthieu Morel. On hierarchical, parallel and distributed components for Grid programming. In V. Getov and T. Kielmann, editors, *Workshop on component Models and Systems for Grid Applications*, ICS '04, Saint-Malo, France, June 2004.

[5] F. D. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96: Proc. of the 1996 ACM/IEEE Conf. on Supercomputing (CDROM)*, page 39, 1996.

[6] J. Buisson, F. André, and J.-L. Pazat. Dynamic adaptation for grid computing. In *European Grid Conference 2005*, Amsterdam, February 2005.

[7] J. Buisson, F. André, and J.-L. Pazat. Enforcing consistency during the adaptation of a parallel component. In *The 4th Int.l Symposium on Parallel and Distributed Computing*, July 2005.

[8] Brian Ensink, Joel Stanley, and Vikram Adve. Program control language: a programming language for adaptive distributed applications. *Journal of Parallel and Distributed Computing*, 63(11):1082–1104, November 2003.

[9] Malcolm McIlhagga, Ann Light, and Ian Wakeman. Towards a design methodology for adaptive applications. In *Mobile Computing and Networking*, pages 133–144, May 1998.

[10] S. Vadhiyar and J. Dongarra. Self adaptability in grid computing. *International Journal Computation and Currency: Practice and Experience*, 2005. To appear.

[11] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.